Project report
Jet Propulsion Laboratory Summer Internship Program
Student–Faculty Programs, California Institute of Technology

NASA's Advanced Multimission Operations System: A Case Study in Formalizing Software Architecture Evolution

Jeffrey M. Barnes (jmbarnes@cs.cmu.edu)
MENTOR: Brian Giovannoni (Brian.J.Giovannoni@jpl.nasa.gov)
CO-MENTOR: Oleg Sindiy (Oleg.V.Sindiy@jpl.nasa.gov)

2011 August 15

ABSTRACT: All software systems of significant size and longevity eventually undergo changes to their basic architectural structure. Such changes may be prompted by evolving requirements, changing technology, or other reasons. Whatever the cause, software architecture evolution is commonplace in real-world software projects. Recently, software architecture researchers have begun to study this phenomenon in depth. However, this work has suffered from problems of validation; research in this area has tended to make heavy use of toy examples and hypothetical scenarios and has not been well supported by real-world examples. To help address this problem, I describe an ongoing effort at the Jet Propulsion Laboratory to re-architect the Advanced Multimission Operations System (AMMOS), which is used to operate NASA's deep-space and astrophysics missions. Based on examination of project documents and interviews with project personnel, I describe the goals and approach of this evolution effort and then present models that capture some of the key architectural changes. Finally, I demonstrate how approaches and formal methods from my previous research in architecture evolution may be applied to this evolution, while using languages and tools already in place at the Jet Propulsion Laboratory.

C O N T E N T S

1 INTRODUCTION

The discipline of software architecture, which deals with the high-level design of software systems, has in recent decades become widely accepted as an essential means of designing software systems effectively. However, one topic that existing approaches to software architecture do not address well is *software architecture evolution*. Software architecture evolution is a phenomenon that occurs in all software systems of significant size and longevity. As a software system ages, it often needs to be structurally redesigned to accommodate new requirements, new technologies, or changing market conditions. In addition, many systems over the years tend to accrue a patchwork of architectural workarounds, makeshift adapters, and other forms of architectural detritus that compromise the architectural integrity and maintainability of the system, requiring some sort of architectural overhaul to address. At present, however, software architects have few tools to help them plan and carry out these kinds of architecture evolution. While there is a sizeable body of research literature on software maintenance and evolution generally, little work has been devoted to this common problem of architecture evolution.

In my research as a doctoral student at Carnegie Mellon University, I have attempted to address this gap in the research. Along with my research advisor and other colleagues, I have been studying the phenomenon of software architecture evolution and developing methods and models to support architects in planning such evolutions [7]. Our approach is based on capturing architectural expertise about classes of evolutions and developing tools to facilitate reuse of this expertise.

One significant obstacle in our research thus far, as well as that of other researchers working in the same area [21, 22, 26], has been the challenge of validation. Although we have made considerable headway in the task of developing a theory of architecture evolution, finding real-world cases on which to test our model has been a challenge. Most of our work so far, therefore, has been based on toy examples, artificial evolutions in laboratory conditions, and low-fidelity formalizations of real-world examples of which we have a general description but no detailed knowledge.

This summer, during a ten-week internship at NASA's Jet Propulsion Laboratory (JPL), I undertook a case study of software architecture evolution. JPL is currently in the process of carrying out a major evolution of an important software system. By examining project documents and speaking with project personnel, I gained an understanding of this evolution. Then, I applied the methods we had developed in our research to the problem of analyzing certain aspects of this evolution. I constructed a formal model of the evolution plan and demonstrated how our research approach could be applied in a real-world engineering context, while using languages and tools already in place at JPL. In this report, I give an overview of this work.

Section 2 describes our approach to software architecture evolution in greater detail and then discusses the context of evolution at JPL. Section 3 describes the specific evolution that I modeled during my internship. Section 4 explains how I applied our evolution analysis approach to this evolution. Section 5 summarizes the results of this effort. Section 6 concludes by considering what lessons may be drawn from the case study. Note also that a glossary of abbreviations is provided as an appendix on page 19.

2 BACKGROUND

2.1 *Software Architecture*

Software architecture is the subdiscipline of software engineering that pertains to the high-level design of software systems. In the last fifteen years or so, a great deal of progress has been made in developing techniques that help software engineers to effectively design software systems to meet functional and quality goals; to document software systems in a way that is useful to their stakeholders; and to apply formal methods (i.e., techniques grounded in mathematics and formal logic) to the analysis of software architectures to understand their properties.

Software architects represent software systems in terms of the high-level elements from which they are made, often using *architecture description languages* specifically designed for representing software architectures. The most important kinds of architectural elements are *components* (the computational elements and data stores of a system) and *connectors* (the interaction pathways among components) [2]. At the most basic level, a software architecture can be thought of as a graph (in the sense of graph theory) in which the nodes are components and the vertices are connectors. In practice, architectural descriptions are made considerably more complex by the addition of other kinds of architectural elements (such as *ports*, which are the interfaces by which one may interact with a component); the annotation of architectural elements with properties to facilitate analysis; and other intricacies. In addition, software architects often consider a system from multiple *viewpoints*—not only the run-time component-and-connector view, but also (for example) views of the code structure and the physical deployment of software onto hardware.

Although software architecture is a rather young field, one which is still maturing, it has grown rapidly in importance and influence. Aside from being an active area of research, software architecture is practiced in some form at nearly all real-world software organizations of significant size.

2.2 *Software Architecture Evolution*

While representing software architectures has been a fairly well understood problem for some time now, the problem of understanding software architecture evolution is just beginning to be explored. At Carnegie Mellon University, we have developed an approach to understanding and modeling software architecture evolution, supported by formal methods that lend themselves to automation [7].

We begin with the assumption that there is a known start state and a known end state. That is, we assume that the current state of the architecture is known, as is the architecture to which the organization would like to evolve. In practice, these assumptions may not hold. However, other research areas address the problems of determining the architecture of an existing system and designing an architecture for a future system (architecture reconstruction and architectural design, respectively), so we do not address them. Instead, we are concerned with how to get from the current state to the target state.

We start by contemplating the set of potential *intermediate states* between the initial architecture and the target architecture—the transitional states that the system may assume as it evolves from its initial form to its target architecture. We represent all the intermediate states, together with the initial and target states, as nodes in a directed graph. We then draw an edge from node *a* to node *b* if there is an evolutionary transition from state *a* to state *b*. In addition, we allow nodes and edges to be annotated with an extensible set of architectural

properties that further characterize the evolution. These properties support analysis of the evolution.

This conceptual setup is fairly simple, but it accomplishes a few things. First, it allows us to see the different ways that a system can evolve. In particular, it allows us to consider the set of possible *evolution paths*—complete routes from the initial state to the target state—and consider tradeoffs among them. We can also visualize things like release points and milestones by demarcating them with node properties. This setup is also amenable to various kinds of analysis.

One of the simplest kinds of analysis that can be done is the analysis of which evolution paths are possible, or legal. One of the basic elements of our model of architecture evolution, therefore, is the notion of a *path constraint*, which is an analysis indicating which paths are legal with respect to some rule about the evolution domain. Formally, a path constraint is a predicate over evolution paths; for each path, a constraint either allows it or forbids it. An example of a constraint might be: "Once a component is migrated to a data center, it must remain in that data center for the rest of the evolution." For any given evolution path, this constraint will either hold or fail to hold.

Another important class of analyses is *path evaluation functions*. While a path constraint provides a hard, yes-or-no judgment about the *legality* of a path, a path evaluation function instead provides a quantitative assessment about the *goodness* of a path, for example by estimating its duration or cost.

These concepts are fairly simple, but there is a substantial formal framework supporting them. To formalize path constraints, for example, we have developed a formal language based on linear temporal logic. This sort of application of formal methods has several advantages, the most important of which are precision (by using a formal approach, we minimize ambiguity, which helps to pin down exactly what project stakeholders mean when they talk about the project) and automation (when we use a formal approach, we can develop tools to make it easier to plan and analyze the evolution).

2.3 Architecture Evolution at JPL

Architecture evolution of this kind is particularly common and important at JPL. Here, missions may last many decades, and software systems must evolve to support them continuously. The Voyager mission launched in 1977, and 34 years later it is still running—and transmitting telemetry that must continue to be processed by software on the ground. The flight software and ground software associated with this mission have required continuous maintenance to keep them up and running for 34 continuous years. This maintenance entails not only routine collection and analysis of telemetry, but also occasional software evolution as well as responses to sporadic anomalies, as in 2010 when a flight software glitch left Voyager 2 nonfunctional until project engineers could repair the software, allowing it to continue reporting on its journey out of our solar system [3]. The Voyager probes are expected to continue transmitting telemetry at least until 2025, when they will at last have insufficient power to support any of their instruments, for a total mission length of nearly half a century [14].

But long mission durations are only one reason that software evolution is so important to JPL. Perhaps even more significant are the *multimission* software systems that JPL maintains. Today, JPL is constantly maintaining the software for a wide array of missions; at the time of this writing, JPL currently has 87 active missions, according to its website—from the Active Cavity Irradiance Monitor Satellite to the Wide-Field Infrared Survey Explorer [13]. Each of these missions has plenty of custom software written for it, but most of them also make use of multimission software—software that is shared among several missions.

JPL takes a sort of product line approach to multimission software; it develops software for multimission use, then adapts it for each mission. Of course, multimission software lasts longer than a typical single mission, and it also has greater evolution needs. As new missions make use of a multimission platform, the platform must evolve to support the new capabilities and qualities that the new missions require. Over a long period of time, a multimission system can change drastically, ultimately to the point where it bears little resemblance to its ancestral form.

The best example of such a multimission system at JPL is the Advanced Multimission Operations System (AMMOS), the title character of this report. AMMOS is the ground software system used for JPL's deep-space and astrophysics missions [11]. It was developed beginning in 1985, with the goal of providing a common platform to allow mission operators to manage ground systems at lower cost than would be possible by building mission-specific tools, without compromising reliability or performance [8]. The system has been used for many prominent NASA missions, and continues to be used today [15].

Architecturally, AMMOS is a *system of systems*; although it functions as a coherent whole with a common purpose, it is composed of disparate elements, each of which has its own engineers, its own users, and its own architectural style. Among the systems that make up AMMOS are elements responsible for uplink and downlink of spacecraft telemetry, for planning command sequences, for processing spacecraft telemetry, for navigation, and so on [11].

AMMOS has served JPL well for many missions, but it is an aging system, and the limitations of its architecture are now becoming apparent [16]. The architecture is resistant to evolution and expensive to maintain. The current system suffers from architectural inconsistencies and redundancies and lacks a coherent overarching architecture. Requirements changes often necessitate modifications that span many subsystems, and the system relies on large amounts of "glue" code—adapters and bridges connecting different parts of the system in an ad hoc way that makes maintenance difficult.

Now, ongoing architecture modernization efforts aim to address this situation by re-architecting AMMOS in a way that makes use of modern architectural styles and patterns [16, 23]. This will allow easier, less expensive maintenance and evolution of AMMOS in the future and also facilitate easier customization of AMMOS for individual missions. The goal is to develop a modern deep-space information systems architecture based on principles of composability, interoperability, and architectural consistency.

3 EVOLUTION DESCRIPTION

In this project, I focused on one element of AMMOS that is of particular interest from the standpoint of software architecture evolution. Confining my work to the evolution of a single AMMOS element over a relatively brief interval of time, rather than attempting to capture a broader view of the evolution of AMMOS, was necessary to achieve my goals within the ten-week duration of my internship at JPL. To achieve a full understanding of the evolution, develop a meaningful evolution description, and produce compelling evolution analyses would not have been possible had I selected a broader scope.

The AMMOS element on which I chose to focus is the one responsible for mission control, data management and accountability, and spacecraft analysis (MDAS) [12]. The MDAS element was an attractive choice for several reasons. First, it was undergoing a major restructuring to meet specific goals. Second, it had an explicit initial software architecture, and the target architecture was also reasonably well understood. Third, it presented interesting tradeoffs and unanswered questions that might be usefully addressed by an architecture

evolution analysis. Fourth, I had good access to staff who were familiar with the system and with the evolution, who could provide architectural information beyond that available in official documentation.

The MDAS element has a number of responsibilities, but one of the most important is to process, store, and display telemetry and other mission data from deep-space operations [12]. Prior to the Mars Science Laboratory (MSL) mission, this responsibility was fulfilled by an assortment of different subsystems, including the Data Monitor and Display (DMD) assembly; the Tracking, Telemetry, and Command (TT&C) system; and a number of others [10]. For the MSL mission, a new system was developed to supplant this complex of systems: the Mission Data Processing and Control System (MPCS) [6].

MPCS was originally developed as a testing platform modeled after the ground data systems for the Mars Exploration Rovers; later it was promoted to support operations for MSL [6]. Engineers are now adapting and refining the architecture of MPCS for multimission use. In this section, I describe the current MPCS architecture (as used by MSL), the motivations for evolving it, and the planned future architecture.

3.1 Initial Architecture

MPCS has an event-driven message bus architecture. All the major components of the system communicate via a Java Message Service (JMS) message bus [6]. JMS is a programming interface that provides for reliable, asynchronous communication among software systems [9]. This promotes loose coupling of software components without compromising reliability. Components can be attached to or detached from the message bus freely (by subscribing to or publishing the appropriate kind of event), provided that they adhere to application protocols and do not violate architectural constraints, allowing for plug-and-play reconfiguration of the system. The components are Java-based and platform-independent; the interfaces by which they communicate are based on XML [6].

This event-driven, bus-mediated architecture gives MPCS a degree of architectural flexibility. That is, there is not really any one "MPCS architecture;" rather, MPCS can be configured in different ways to achieve different goals. At its most flexible, MPCS can be regarded a loose confederation of tools rather than a cohesive system with a fixed design. However, MPCS does have a rather stable infrastructure of core components that are generally connected in a well-defined way, so for most purposes we can treat MPCS as a system with a stable platform and a fixed set of variation points, and indeed this is how we will treat MPCS here.

An important example of the architectural variability of MPCS is that it is deployed with quite different configurations in different environments. MPCS is used in several environments—not only mission operations, but also flight software development; system integration; and assembly, test, and launch operations (ATLO)—and there are significant differences in architectural configuration among the environments [6]. For flight software development, for example, MPCS can be used to issue commands to the flight software under development; in operations, however, commanding features are delegated to a different system called CMD, which is external to MPCS and indeed external to AMMOS (it is a subsystem within JPL's Deep Space Network).

The most important components of MPCS are:

- The aforementioned **message bus**.
- The telemetry processing subsystem of MPCS, which is called **chill_down** [1] (*chill* is a code name for MPCS [24], and *down* is for *downlink*). This component takes as input an unprocessed telemetry stream from a spacecraft (or other telemetry source, such as a simulation environment), performs frame synchronization and

packet extraction, and processes packets to produce event verification records and data products [1].

- The commanding component of MPCS, which is called **chill_up** (*up* for *uplink*) [4, 5]. This component transmits commands to the flight software (or simulation environment). Currently, chill_up is used only in the flight software development and ATLO environments, not in operations.
- The **MPCS database**, a MySQL database, used for storing telemetry as well as some other information, such as logs and commanding data [5]. This database is queried by a number of analysis components.
- The monitoring interface, called **chill_monitor**, used for real-time display of telemetry [4, 5]. There are generally many instances of chill_monitor for a single instance of MPCS, as many mission operators may be monitoring telemetry simultaneously.
- A variety of **MPCS query** components, with names like *chill_get_frames*, *chill_get_packets*, *chill_get_products*, and so on [5]. These programs retrieve data from the database and output the data in a standard format.

Together, these components effectively form a standard MPCS workflow. Commands are issued by chill_up and conveyed to the flight software (or simulation environment), which carries out the commands; the flight software produces telemetry, which is processed by chill_down. The chill_down component stores the processed telemetry to the MPCS database (where it is queried by the MPCS query components) and transmits messages about the processed telemetry to the message bus (where it is displayed by chill_monitor). Although MPCS is flexible enough to be configured in many different ways, this workflow describes the way MPCS works most of the time, in typical environments. In section 5, I will show an architectural diagram of MPCS illustrating these components and how they connect.

3.2 Impetus for Evolution

It is believed that MPCS will serve adequately for the MSL mission. However, as MPCS is developed for reuse in future missions, engineers face the need to evolve the system to improve qualities such as performance and usability, support additional needed capabilities, and better integrate with other ground data systems. In my work this summer, I focused on two particular proposed features of MPCS that project architects hoped to introduce in future versions: *integrated commanding (ICMD)* and *timeline integration*.

ICMD is motivated by the NASA principle "test like you fly." That is, NASA aims to make system-testing environments as similar as possible to actual spaceflight operations. As we have seen, one of the most salient architectural characteristics of MPCS is that it takes different forms in different environments. In particular, there are important architectural differences between the testing environment (ATLO) and the spaceflight operations environment. The ICMD effort aims to bring the operations environment more in line with the ATLO environment.

The main difference between the testing configuration of MPCS and the operations configuration of MPCS is commanding. In ATLO, the chill_up component of MPCS is responsible for issuing commands to the spacecraft. In operations, the responsibility of issuing commands is excised from MPCS entirely; instead, the CMD system is responsible for issuing commands. ICMD will change the operations environment to look more like ATLO; the chill_up component of MPCS will now issue commands in all environments.

Timelines are a new data structure proposed for storing streams of time-oriented data throughout AMMOS. A "timeline" is exactly that: a linear sequence of events with associated times, in chronological order. Many of the kinds of data that JPL handles on a day-to-day basis fit naturally into this model: telemetry, command sequences, and others. The

timeline proposal defines specific formats for storage and transmission of timelines, and also describes the architectural infrastructure necessary to support them. Timelines are expected to be useful for many purposes, but one of the most important is comparison of actual telemetry with expected telemetry. Mission operators need this capability all the time (comparing an observation with a theoretical prediction is one of the most basic requirements in science), but currently comparing expected and actual telemetry is a manual, laborious operation. Supporting timelines will require substantial architectural infrastructure. Although the basic idea is not complex, there are very stringent performance requirements; processing timelines must be very fast. Thus, for example, a specially engineered timeline database is planned, which will be designed specifically for efficient storage and retrieval of timelines.

The introduction of timelines will have ramifications for many of the AMMOS elements, including MPCS. Currently, MPCS stores telemetry information in a MySQL database. Ultimately, this database will be rendered obsolete by the introduction of timelines. After timelines are integrated into MPCS, telemetry will be stored in an AMMOS-wide timeline management system, and the MySQL database will eventually be retired. Other parts of MPCS will also be affected by the introduction of timelines; for example, the subsytem for mission planning and sequencing is likely to see changes as well.

3.3 Target Architecture

The main differences between the end state of the evolution and the initial state are greater homogeneity among deployment environments (supporting the "test like you fly" philosophy) and integration with the new timeline infrastructure (improving on the usability of the current architecture).

In the end state, chill_up will be used for commanding in all environments, including spaceflight operations. The CMD system will continue to exist, but will no longer be the originator of commands in the operations environment. Instead, chill_up will originate commands and convey them to CMD, which will prepare them for uplink.

More precisely, chill_up will not directly send commands to CMD, but will instead transmit *references* to commands that it has stored in a command repository; CMD will then access this repository to read the actual commands. Once timelines are introduced, this command repository will become obsolete, as command sequences will be stored as timelines and will therefore be stored by the timeline management system. This is an important point of interaction between the two pieces of our evolution, ICMD and timelines.

In addition to storing commands, the timeline management system in the end state will be responsible for storing channelized telemetry. Thus, the target architecture lacks the MySQL database that exists in the initial architecture, and the usages of that database by other MPCS components are replaced by connections with the timeline management system. The timeline management system will be external to MPCS, so these connections will be external collaborations rather than internal connections.

4 APPROACH

I spent the early weeks of my internship gathering information. My first goal was to familiarize myself with the particulars of the various elements of AMMOS and the plans that were in place for evolving them. I did this by reviewing project documents and speaking with project personnel.

My second goal was to select an evolution to study. At the beginning of my project, we had not yet selected the specific evolution that I described in section 3. On the contrary, a

diverse array of options was under consideration. I worked together with my mentor and project contact at JPL to select an appropriate evolution. Among the options were past evolutions (i.e., those that had already been finished and whose outcome was therefore known); current evolutions (i.e., those which were ongoing); and future evolutions (i.e., those that were under consideration to occur in the future). We ultimately selected an evolution that was in progress but had only been underway for a short time. Picking a current evolution had the advantage of being of greatest relevance to JPL. Another advantage was that there were ample resources for learning about the evolution; it was easy to find project personnel who could share accurate, timely information about evolution plans. If we had selected a past evolution, it is likely that documentation would have been difficult to find, and there would have been few current personnel who were familiar enough with the evolution to provide useful information. On the other hand, if we had selected a future evolution for which few firm plans had been made, we would have had to engage in substantial speculation to construct an evolution graph.

Another important choice was the scope of the evolution, in terms of both time (i.e., a long evolution versus a short one) and breadth (i.e., the evolution of a small subsystem versus the evolution of a large chunk of AMMOS). We could have picked a much larger scope than we did—for example, by studying the overall evolution of AMMOS rather than focusing on MPCS, or by trying to look further into the future. However, given the short duration of my internship, it would have been difficult to gather sufficient information about a broader evolution to produce a useful model capable of saying anything useful about the evolution—one that was more than a superficial overview. A more narrowly scoped study, on the other hand, would have shown changes that were too minor to be interesting.

Once I had selected an evolution to study, the next task was to model it. In previous work at Carnegie Mellon University, we have modeled architecture evolution using research languages and research tools. One of my aims in this project was to evaluate the practicality of implementing our approach to architecture evolution analysis while using commercial languages and tools—in particular, those already in use by the software organization under study. At JPL, the dominant language used for modeling systems is SysML, the Systems Modeling Language, and the dominant modeling tool is MagicDraw, a commercial tool that can produce SysML models. In the rest of this section, I will describe SysML and MagicDraw and explain how I adapted our approach to architecture evolution to this environment.

4.1 SysML

SysML is a specialization of an older and better-known modeling language called UML (the Unified Modeling Language), which was developed for modeling software systems. UML, originally standardized in 1997 and revised many times since, is a general-purpose modeling language that has proved very popular in the software engineering community. UML provides an array of different *diagram types*, which allow a software system to be represented from different perspectives—for example class diagrams (used for describing the structure of a software system's source code), deployment diagrams (used for describing the hardware used by a software system), and sequence diagrams (used for describing how objects in a software system communicate).

SysML is a specialization of UML for the domain of systems engineering (rather than software engineering specifically). SysML is formally defined as an extension, or *profile*, of UML. SysML arose from collaboration, beginning in 2001, between the Object Management Group (the standards consortium behind UML) and the International Council on Systems Engineering (a professional organization of system engineers). In 2003, the Object Management Group issued a Request for Proposal for a customization of UML for systems engineer-

ing [20]. A coalition of industry leaders formed and developed an open-source specification to meet the Request for Proposal [25]. In 2006, the Object Management Group adopted SysML as a standard [18]. SysML has since undergone minor revisions; the latest version is SysML 1.2, released in 2010 [19].

SysML is simultaneously a restriction and extension of UML. It is an extension in the sense that it adds new syntax and semantics beyond that defined in UML. It is a restriction in that it excludes many of the elements that do exist in UML, for the purpose of simplifying the language. SysML takes a subset of the diagram types from UML and repurposes them for the domain of systems engineering. The class diagrams of UML, for example, become *block definition diagrams* (BDDs) in SysML; composite structure diagrams become *internal block diagrams* (IBDs).

4.2 Why SysML instead of UML?

UML is used for modeling software systems, while SysML is a profile of UML that is intended for systems engineering applications; since I was modeling software systems, UML might seem a more natural choice for this project than SysML. In addition, the tools that I used for this project supported UML as well as SysML, so using UML would have been technically feasible. I did consider using UML, but there were several reasons I settled on SysML.

First, SysML was already heavily used within the organization I worked in at JPL (and throughout JPL more broadly), including for modeling software systems. Many engineers were already comfortable with SysML and considered it a suitable language for modeling systems even when those systems had few or no nonsoftware components. By using SysML, I hoped to make my work more readily accessible and potentially useful to a JPL audience.

Second, SysML has certain metamodel extensions whose semantics are particularly well suited to the software systems I was modeling. In particular, SysML introduces *flow ports* and *item flows*, which are ports and connectors through which items may flow. (In the context of AMMOS, we are principally interested in modeling the flow of *data*. In systems engineering contexts, these modeling constructs may represent literal, physical flows, such as flows of fluid or energy.) These constructs provide a natural way of modeling how information flows through a software system. Since data flow was a dominant way of understanding the functioning of some portions of MPCS, it seemed that flow ports and item flows would be useful for my modeling task. For example, much of the communication in MPCS occurs asynchronously via a message bus; SysML flow ports and item flows are well suited to capturing the semantics of this kind of communication.

Third, one of the appeals of SysML is that it is a deliberate simplification of UML. In addition to providing its own, new metamodel elements, it also reduces the number of available UML metamodel elements. This simplifies the modeling process by reducing the number of choices available. The elements available in SysML would be more than sufficient for my purposes.

Ultimately, the choice between UML and SysML had little practical effect; the project could have proceeded in essentially the same way had I used UML. I could have used the same tool (MagicDraw), the same approach to modeling the systems under study, the same constraint language (OCL), and so on. Indeed, SysML and UML are so closely related that the architectural representations themselves might have looked much the same; SysML BDDs could have been UML component diagrams, and SysML IBDs could have been UML composite structure diagrams. In some respects, the choice between UML and SysML was thus an academic one.

4.3 *Representing Software Architecture in SysML*

SysML was not designed to serve as a formal software architecture description language, so I had to establish some conventions so that it could serve as one. I chiefly used two diagram types for the entire project: BDDs and IBDs. (I also used one package diagram, but we will come to that later.) In SysML, a *block* is the basic unit of system structure. A BDD shows the blocks that appear in the model; an IBD shows the internal structure of a block. I used BDDs to show the kinds of architectural components in my model and the hierarchical relationships among them; I used IBDs essentially as conventional software architectural diagrams, to show the architectural structure of a system. BDDs and IBDs are both representations of an underlying model.

I tailored my use of the diagram types to show those aspects of the architecture whose evolution we hoped to model. The most detailed and important diagram that I produced was an IBD showing the internal structure of MPCS. I also produced a set of three IBDs that served essentially as context diagrams showing how MPCS was deployed in the three different environments (flight software development, ATLO, and operations). For representing this evolution, it was important to see not only the internal changes that were occurring within MPCS, but also the changes in how MPCS interacted with other systems, such as CMD. Recall that such changes were key to the overall evolution, so modeling them was crucial to providing a complete, useful representation of the evolution.

Although my use of SysML was guided by conventions that I established for myself to match a particular conception of software architecture, it is important to emphasize that I remained faithful to the semantics and conventions of SysML. That is, I did not wantonly abuse the language to fit a particular paradigm; instead, I made use of those elements of the language which were already compatible with that paradigm. In fact, my use of block definition diagrams and internal block diagrams is similar to examples in the SysML specification [19]. See particularly section B.4.5 of the specification: block definition diagrams such as figure B.18 show the kinds of components that appear in the system, while internal block diagrams such as figure B.19 show the details of how these pieces fit together to form the architecture of the system (cf. figures 1 and 2 in this report).

Many diagrams of the initial system (MPCS as it existed in MSL) already existed before I began my internship—including one representation of the system in MagicDraw with SysML. (Some of these diagrams are publicly available; see, e.g., [4–6].) I studied these diagrams carefully but ultimately decided to create my own diagram rather than reusing any of these. There were a few reasons for this. First, in creating my own diagram, I could ensure that it was consistent with formal software architecture principles and amenable to the kinds of analysis that I wanted to do. Second, there were a number of inconsistencies among the existing diagrams, and creating my own diagram from scratch was simpler than attempting to account for all these discrepancies. Third, my representation of MPCS would differ from previous diagrams in ways that were important for technical reasons, as we will see in section 5.1.

4.4 *Modeling Software Architecture Evolution with MagicDraw*

Modeling software architecture in SysML is fairly straightforward; after all, SysML is a profile of UML, which is specifically designed for software architecture representation. More interesting is the question of how to model architecture *evolution* effectively. Recall that we model an evolution as a graph, in which the nodes are intermediate architectures and the edges are transitions. The first step in representing an architecture evolution, then, is figuring out how to represent the nodes. Of course, the simplest strategy would be to create one MagicDraw project for each intermediate state. A better idea, however, is to include all the

intermediate states, and hence the entire evolution graph, in a single project. With everything in one project, it becomes possible to write evolution constraints and analyses with existing tools, simply by using the model constraint and analysis facilities already provided by the tool.

More specifically, I placed each intermediate state in its own package. A package is a UML construct (also available in SysML) that encapsulates related entities. Placing the intermediate states into different packages allows them to be as isolated as necessary, while still existing within the same project so as to accommodate analyses of the entire evolution graph. In addition, we can represent the packages themselves in a *package diagram*; then we can represent the transitions between them by relationships among the packages. Finally, if we wish, we can add annotate these packages and relationships with additional information to facilitate analysis, such as node properties and edge properties.

4.5 Representing Model Transformations

The problem with having a separate representation for each intermediate state is that it can be a maintenance nightmare. Since what I was modeling was a gradual evolution rather than an outright retirement and replacement of a system, all the states look mostly the same, except for those pieces that are evolving. Thus, modeling the evolution graph required me to produce many nearly identical packages. I could have created this evolution graph model very easily by simply cloning the initial state and modifying it. That is, after first representing the initial state, I could have copied it, pasted it, and modified it to create the next state; then done likewise for the next state; and so on. But maintaining this evolution graph would have been painful.

Suppose that after I had finished representing all the states, I had noticed a mistake in the initial architecture that affected all the other states as well (since they were generated by cloning the initial state). To address the problem, I would have had to fix each state by hand. In the evolution graph I ultimately produced, there were seven states; a more broadly scoped evolution could have many more. Thus, fixing problems in this way would be laborious.

Instead of this copy-and-paste approach, I decided to model the structural transformations themselves in such a way that they could be applied automatically. Rather than generating intermediate states by hand and applying the evolution steps by hand, I would specify the structural transformations needed to generate the intermediate states automatically. Then, if the initial state changes, the intermediate states can be regenerated instantly, so fixes need only be applied in one place instead of many. This is analogous to the way that most revision control systems use delta encoding to store file versions (storing diffs between versions rather than a complete copy of every version of every file) or to the way that video compression works (by storing differences between frames, taking advantage of the typical similarity of nearby frames, rather than storing a complete copy of every frame).

This approach accords well with the model of architecture evolution I described in section 2. There, I discussed architectural operations, which capture the structural transformations involved in evolution steps, as well as other information to support analysis. The transformations that I employed in this project fulfilled the same role here (except without providing metadata to support analysis).

I implemented these transformation specifications as macros in MagicDraw. MagicDraw supports the definition of fairly sophisticated macros that can alter both the model and the presentation of its diagrams. To do so, it exposes a rich Java API for creating and modifying model elements and presentation elements. Macros are written in a scripting language and compiled to Java bytecode.

There are several languages in which MagicDraw macros can be written: Groovy, BeanShell, JavaScript, JRuby, and Jython. All of these are dynamically typed programming languages that can compile to Java bytecode and run on the Java platform, so the choice among them is largely one of personal preference. I selected Groovy, whose syntax is based on Java but is rather laxer (e.g., semicolons and type declarations are unnecessary) and also introduces many additional features (e.g., for functional programming).

In principle, one could use a UML transformation standard such as QVT for this purpose, rather than a script using a proprietary API like MagicDraw's. For this project, I decided to use macros rather than QVT for two reasons. First, MagicDraw has no built-in support for QVT (nor any other model transformation language), and although there is an official MagicDraw plug-in for QVT, it is immature, somewhat buggy, and not well documented. Second, using macros allowed me to transform not only the model, but also the diagrammatic presentation of that model. With QVT I would have been limited to the former; I could have transformed the model automatically, but still would have had to update the diagrams by hand, eliminating much of the benefit of the automated approach.

4.6 Modeling Constraints and Analyses with OCL

Constraints and analyses are important parts of our model of software architecture evolution. Because of the short duration of my internship, I did not have an opportunity to develop a rich variety of constraints and analyses for the evolution that I studied. However, with the entire evolution graph represented in a single model, it should be possible to represent constraints and analyses as OCL constraints over the model.

OCL, the Object Constraint Language, is a declarative language for specifying rules of models in UML (and other modeling languages governed by the same metamodel as UML) [17]. OCL was originally developed to annotate UML models with additional constraints that are inexpressible in UML; however, it can also be used to express constraints over UML models—to judge whether a particular UML model satisfies some constraint. This makes it suitable for expressing evolution constraints, given that our entire evolution graph is in one model.

In principle, I believe that it should even be possible to develop an algorithm for translating, or compiling, constraints in our temporal-logic-based constraint language into OCL. Thus it would be possible to develop, say, a MagicDraw plug-in that allows architects to express constraints in this temporal logic, then transparently compiles them to OCL and checks them against the model. However, I did not have time during my internship to explore this idea.

Of course, macros are an option here too, and might provide some additional flexibility, at the cost of portability. Macros might be particularly useful for expressing evaluation functions, as OCL's constraint-based approach may be too rigid for quantitative analysis of the evolution graph.

5 RESULTS

5.1 Representing the Initial Architecture

Figures 1 and 2 show the most important diagrams from my model of the initial architecture of MPCS. Figure 1 is an IBD depicting the internal structure of MPCS. This is a fairly complicated diagram, but there are a couple of features that are particularly worthy of attention. Note first the major components I mentioned in section 3: the JMS message bus, the chill_up and chill_down (uplink and downlink) components, and so on. This is a very

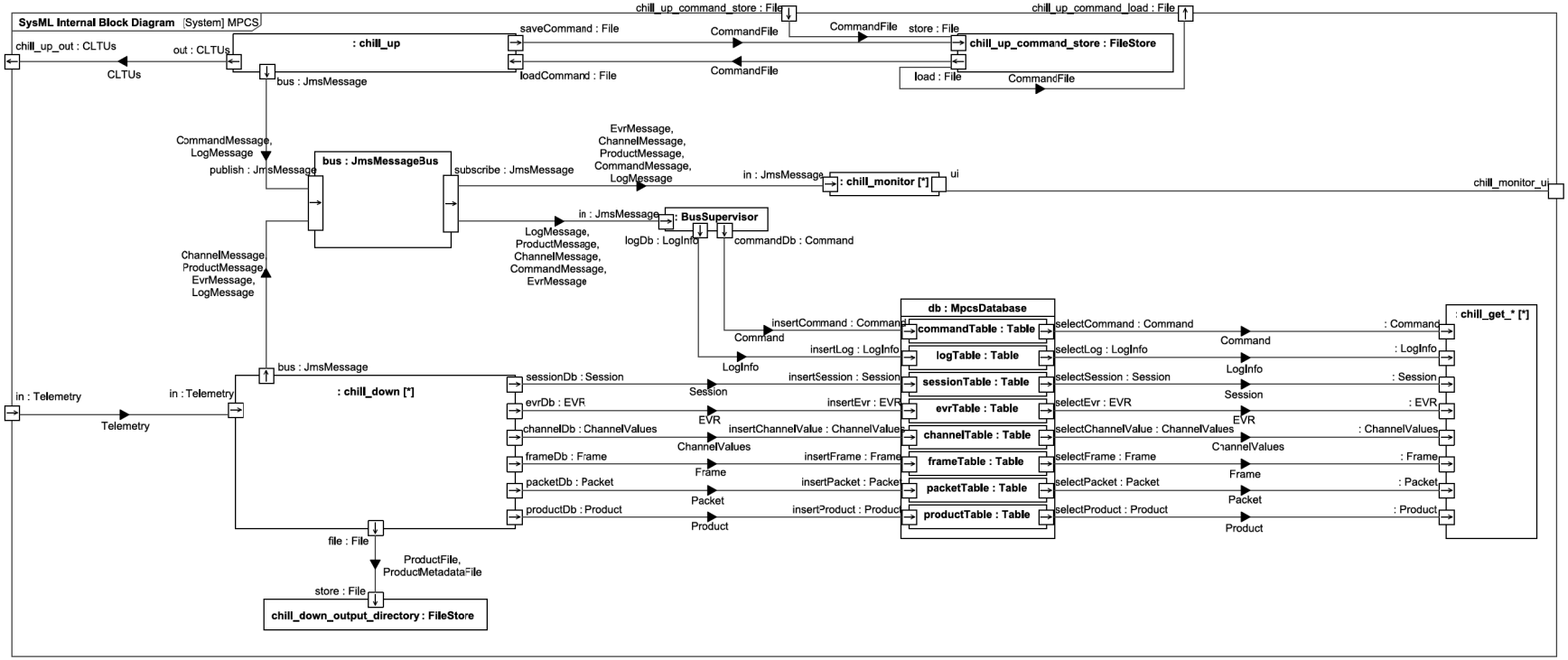
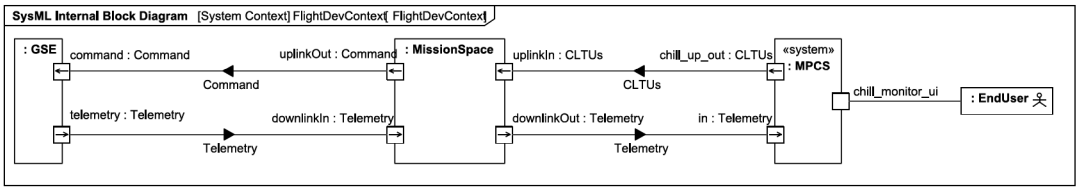
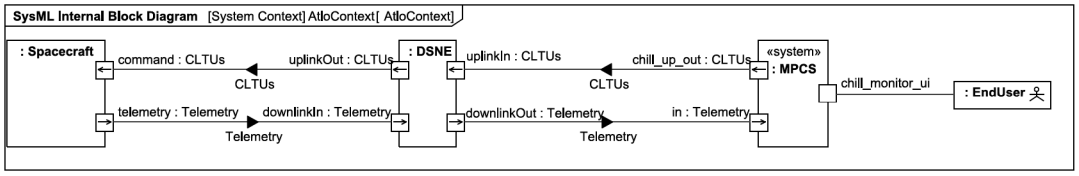


Figure 1: IBD showing the internal structure of MPCS in the initial state

(a)



(b)



(c)

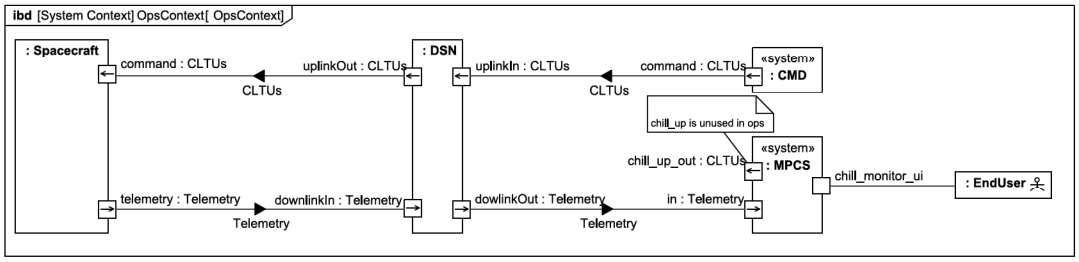


Figure 2: IBDs showing the operational contexts in which MPCS can be deployed: (a) flight development, (b) ATLO, and (c) spaceflight operations

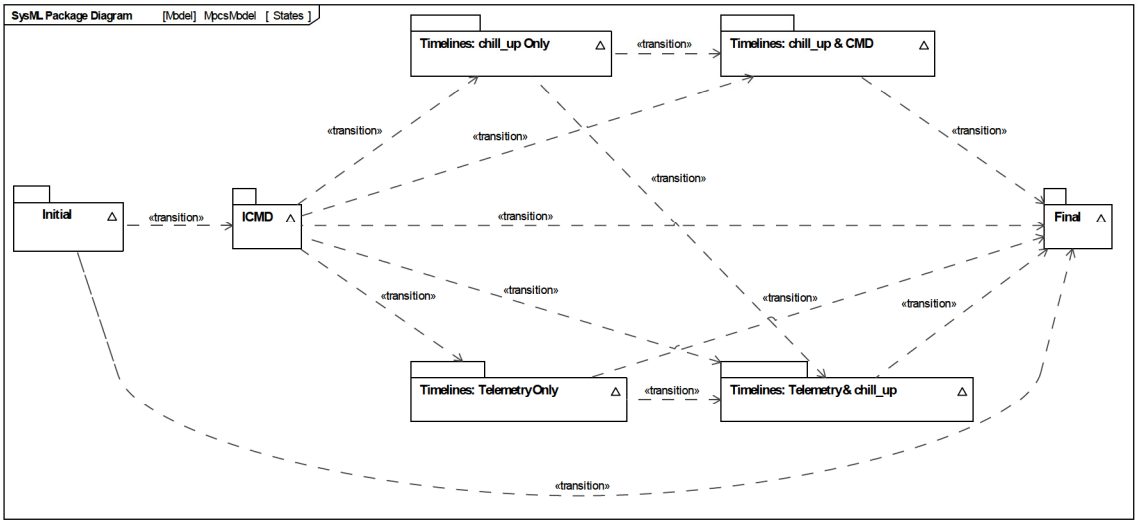


Figure 3: Package diagram showing the evolution graph; states are represented as packages, and transitions are represented as dependencies (dashed lines) among the packages

data-flow-oriented representation of MPCS, which is appropriate given its nature. Most previous architectural representations of MPCS at JPL have also depicted data flow prominently (e.g., [4–6]).

In other ways, however, this representation is quite different from previous representations of MPCS at JPL. The most important difference is that there are some key software elements that previous representations have depicted as being components of MPCS, but that I have represented instead as external collaborators of MPCS. To put it another way, I have drawn the boundary of MPCS quite differently from others at JPL who have represented the system. In particular, previous representations have included inside MPCS components such as MissionSpace, a flight software development simulation environment. I have instead represented MissionSpace, along with all other environment-specific components, as external collaborators of MPCS. This has two advantages. First, it makes it easy to depict MPCS without the difficulty of somehow representing all the different architectural configurations in which MPCS can be deployed. Previous diagrams of MPCS have either addressed this issue by introducing special notation to indicate MPCS components that exist in some environments but not others, or ignored it by tacitly representing only one environment. The second advantage is that these extra components are not really part of MPCS anyway. MissionSpace, for example, is a third-party off-the-shelf tool, and no one thinks of it as being a component of MPCS; previous diagrams have included it as an MPCS component apparently for the sake of convenience and diagrammatic simplicity.

Of course, redrawing the boundary of MPCS in this way does not eliminate the problem of representing multiple environments; it merely pushes the problem outward, so we can deal with it separately. We still do need to represent the different environments, because they feature importantly in the evolution under study (the different architectural configurations evolve differently). Therefore, I produced three more IBDs that show how MPCS interact with its external collaborators (figure 2). Informally, I refer to these as *context diagrams*. Properly, neither UML nor SysML has a context diagram type, so I represented them as IBDs—partial internal representations of the larger system of systems in which MPCS resides (*partial* because I do not include all the ground data system elements, but only the small part of the system that is relevant to MPCS).

These context diagrams show the different environment configurations in a simple way. In subfigure 2a, we see that in the flight software development environment, MPCS both issues commands to and processes telemetry from the simulation equipment. Subfigure 2b shows the AITLO environment, which is mostly the same except that now MPCS is talking to the real spacecraft instead of a simulator. Finally, in subfigure 2c we see the spaceflight operations context, which is different. Here, a separate system is now responsible for commanding, while the uplink port on MPCS is unused.

5.2 Intermediate States and Alternative Paths

I ultimately produced an evolution graph with seven states, including the start and end states. The package diagram in figure 3 shows these states. The mainline evolution path is the simple, two-transition path from the "Initial" state to the "ICMD" state to the "Final" state. The first transition is the introduction of ICMD, and the second is the introduction of timelines. However, a number of alternative paths are possible.

The simplest possible path is to go directly from the initial state to the target state, skipping the ICMD evolution entirely. That is, rather than first integrating MPCS commanding into the spaceflight operations and then integrating timelines, we could go straight to the target architecture. This makes sense because the ICMD and timeline evolutions interact, and in some respects the timeline evolution undoes part of the ICMD evolution. The ICMD

evolution rewires the commanding components of MPCS so that they communicate with the CMD element of the Deep Space Network; the timeline evolution then rewires these same components again so that they can communicate with the timeline management system. As is often the case with evolution paths, there are tradeoffs. Going directly to the target state would be faster and cheaper than going via the ICMD waypoint. However, it would also be riskier—not only because the lack of intermediate releases increases the engineering risk, but also because the lack of stakeholder visibility into the state of the system would increase the risk of project cancellation.

The other alternative paths in this graph emerge during the introduction of timelines. One possibility would be to stage the introduction of timelines instead of introducing them all at once. In particular, the integration of timelines into the uplink portion of MPCS and the integration of timelines into the downlink portion are independent and could be accomplished separately.

Another evolution option has to do with the way that the chill_up component of MPCS interacts with CMD. In the final state, chill_up does not actually send commands directly to CMD; instead, it stores commands to the timeline management system, then passes a references to the command timeline to the commanding element. Instead of integrating timelines into chill_up in this way immediately, we could introduce an intermediate state in which chill_up makes use of the timeline management system itself but continues to send commands to CMD directly.

All of these various possibilities, and the complex interactions between them, appear in figure 3. Behind each of the packages in figure 3 is a complete architectural representation of the system in that state; here, I have shown only one state, the initial state (figures 1 and 2). In the next subsection, I describe how these intermediate-state representations are generated.

5.3 Representing Architectural Transformations

As I said in section 4.5, I used macros to specify the architectural transformations that defined the evolution transitions rather than explicitly specifying each intermediate state by hand. In addition, I limited the size of the transformation macro by building up the transformations out of smaller, reusable pieces. For example, in figure 3, the transition from "ICMD" to "Timelines: chill_up Only" and that from "ICMD" to "Timelines: Telemetry Only" both involve the introduction of a timeline management system, so rather than specify the introduction of the timeline management system twice, I defined it in such a way that it could be referenced by both transitions.

The entire specification for all transformations for the entire evolution graph was on the order of a thousand lines of Groovy code, including transformations of both the model and the presentation of all diagrams. (The exact line count is not yet available, because the final version of the script had not been finished by the deadline of this project report.) The code is reasonably easy to read and write. For example, here is the code that creates the timeline management system block (and its ports):

```
// Create new TMS block in the model
shared.tms = factory.createClassInstance()
shared.tms.name = "TMS"
shared.tms.owner = modelRoot
StereotypesHelper.addStereotypeByString shared.tms, "System"

// Add input port to TMS
shared.tmsInPort = factory.createPortInstance()
shared.tmsInPort.owner = shared.tms
```

```
shared.tmsInPort.name = "in"
StereotypesHelper.addStereotypeByString shared.tmsInPort, "FlowPort"
SysMLHelper.setDirectionFlowPort shared.tmsInPort, "in"

// Add output port to TMS
shared.tmsOutPort = factory.createPortInstance()
shared.tmsOutPort.owner = shared.tms
shared.tmsOutPort.name = "out"
StereotypesHelper.addStereotypeByString shared.tmsOutPort, "FlowPort"
SysMLHelper.setDirectionFlowPort shared.tmsOutPort, "out"
```

5.4 Constraints and Analyses

The evolution graph in figure 3 has eight potential evolution paths. Formalized, automatically checkable constraints and analyses would be helpful for choosing among these paths. Unfortunately, I did not have time during my internship to explore fully the constraints that are in play in this evolution and the analyses that would be useful for understanding it, nor to formalize them. However, I have considered informally the sorts of analyses that would be helpful here and can speculate about how we might capture them.

Many of the concerns pertaining to the alternative evolution paths appear to be based on *risk*. For example, as I mentioned earlier, the primary argument against evolving directly from the initial system to the target system is that it entails substantial risks. Similar tradeoffs are involved in many of the other evolution paths. Risk is a special kind of quality that merits special consideration in a theory of software architecture evolution. At Carnegie Mellon University, we have also encountered the need for a model of risk in our analysis framework but have not seriously addressed the topic. The question of how best to model risk in analyzing architecture evolution remains for future work. In general terms, though, analyzing risk is likely to entail the construction of some sort of probability model for the evolution, so that we can model the likelihood that various contingencies will occur and the effect that those contingencies will have. There is a great deal of existing work on risk modeling, both for the software industry specifically and also in more general contexts such as economics, which we could draw on to develop a model of risk in software architecture evolution. For now, this remains as future work.

Other prominent concerns about this evolution include time, cost, and collaboration. These are somewhat more straightforward to model. Recall that the transitions in our model are composed out of smaller, more elementary transformations; if we can understand the time and cost properties of these elementary transformations, we can compose them to develop time and cost models for the entire evolution graph. Estimating the time and cost of these elementary transformations is still not trivial, but it is considerably easier than attempting to understand the entire evolution graph at once.

These are all "business" issues rather than technical ones. But there are also technical constraints in play in this evolution, and technical constraints are often simpler to analyze in purely structural terms than business constraints. In this evolution, we might have a constraint that there are always complete pathways by which commands may be uplinked to the spacecraft and telemetry downlinked; if not, there is a bug in the model. This is a simple example, but it illustrates how technical constraints may be easily expressible in structural terms.

6 CONCLUSION

This case study was a successful application of software architecture evolution research to the problem of modeling and analyzing a real-world evolution. This project faced a

number of obstacles, including the short duration of my internship, my initial unfamiliarity with the systems under study, and the use of a commercial modeling tool with which I was unfamiliar. The success of the project in the face of these obstacles suggests that formal modeling of software architecture evolution is not only possible in theory, but also may be easier than expected to put into practice in a real-world setting.

Even though I was using an unmodified, off-the-shelf commercial tool, I was able to implement most of the key elements of our research model of architecture evolution with a modest amount of effort in a limited period of time. With custom tool support, formal software architecture evolution modeling could someday be much easier.

ACKNOWLEDGMENTS

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the JPL Summer Internship Program and the National Aeronautics and Space Administration.

I would like to convey my thanks to the many members of the ground systems engineering section at JPL who provided me with help during this project. I am particularly grateful to Brian Giovannoni, Dave Santo, and Oleg Sindiy for their guidance and support throughout my internship.

GLOSSARY OF ABBREVIATIONS

AMMOS Advanced Multimission Operations System
The mission operations system for NASA's deep-space and astrophysics missions

API Application programming interface
An interface defined by a software system to allow other software to communicate with it

ATLO Assembly, test, and launch operations
The project phase in which a spacecraft is built, tested, and launched into space

BDD Block definition diagram
SysML diagram type that shows blocks and the relationship among them

CMD Commanding system
Subsystem of the Deep Space Network responsible for issuing commands

DMD Data Monitor and Display
Assembly that processed and displayed channelized data in "classic" (pre MSL) MDAS

IBD Internal block diagram
SysML diagram type that shows the internal structure of a block

ICMD Integrated Command
An ongoing evolution that aims to integrate the CMD system with MPCS

JMS Java Message Service
A Java standard supporting message-based communication among software components

JPL Jet Propulsion Laboratory
NASA research and development center operated by the California Institute of Technology

MDAS Mission control, data management and accountability, and spacecraft analysis
AMMOS element that provides for spacecraft monitoring and control

MPCS Mission Data Processing and Control System
MDAS system that processes, stores, and displays telemetry from spacecraft

MSL Mars Science Laboratory
Mars rover mission to launch November 2011 and land August 2012

NASA National Aeronautics and Space Administration
United States agency responsible for space exploration and aeronautics research

OCL Object Constraint Language
Declarative language developed to specify constraints on UML models

QVT Query/View/Transformation
A standard language for defining transformations of UML models

SysML Systems Modeling Language
UML-based modeling language for systems engineering applications

TT&C Tracking, Telemetry, and Command
Legacy system responsible for telemetry operations

UML Unified Modeling Language
Standard general-purpose modeling language for software engineering

XML Extensible Markup Language
Standard format for exchange of structured data

REFERENCES

[1] Cervantes, A. Exploring the use of a test automation framework. In: Proceedings of the IEEE Aerospace Conference. IEEE (2009)

[2] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J. Documenting Software Architecture: Views and Beyond, 2nd edition. Addison-Wesley, Upper Saddle River (2010)

[3] Cook, J.-R. C. Engineers diagnosing Voyager 2 data system. Press Release 2010-151, JPL, Pasadena (2010). http://www.jpl.nasa.gov/news/news.cfm?release=2010-151

[4] Dehghani, N. Mission Data Processing and Control Subsystem (MPCS). Presented at: European Ground System Architecture Workshop (ESAW 2007), Darmstadt, June 13, 2007. http://www.egos.esa.int/export/egos-web/others/Events/Workshop/ESAW-workshop-2007/day2-Session-4-0900-1040/04_Dehghani.pdf

[5] Dehghani, N., Sun, Q., Demore, M. NASA's 2011 Mars Science Laboratory (MSL) & supporting ground data system architecture. Presented at: European Ground System Architecture Workshop (ESAW 2009), Darmstadt, May 5, 2009. http://www.egos.esa.int/export/egos-web/others/Events/Workshop/ESAW-workshop-2009/Day1-Session-1-0920-1055/S01_03_Dehghani.pdf

[6] Dehghani, N., Tankenson, M. A multi-mission event-driven component-based system for support of flight software development, ATLO, and operations first used by the Mars Science Laboratory (MSL) project. In: Proceedings of the AIAA International Conference on Space Operations (SpaceOps 2006). AIAA (2006). http://hdl.handle.net/2014/39852

[7] Garlan, D., Barnes, J.M., Schmerl, B., Celiku, O. Evolution styles: Foundations and tool support for software architecture evolution. In: Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA 2009), pp. 131–140. IEEE (2009)

[8] Green, W.B. Multimission ground data system support of NASA's planetary program. Acta Astronaut. **37**, 407–415 (1995)

[9] Hapner, M., Burridge, R., Sharma, R., Fialli, J., Stout, K.: Java Message Service, version 1.1. Sun Microsystems, Palo (2002). http://download.oracle.com/otndocs/jcp/7195-jms-1.1-fr-spec-oth-JSpec/

[10] Kurtik, S.C., Berner, J.B., Levesque, M.: Calling home in 2003: JPL roadmap to standardized TT&C customer support. In: Proceedings of the International Space Ops 2000 Symposium (2000). http://hdl.handle.net/2014/14262

[11] NASA JPL: AMMOS. http://ammos.jpl.nasa.gov/

[12] NASA JPL: Mission Control Data Management & Spacecraft Analysis (MDAS). http://ammos.jpl.nasa.gov/moremgss/missiondataspacecraftanalysis/

[13] NASA JPL: Missions (2011). http://www.jpl.nasa.gov/missions/

[14] NASA JPL: Voyager: Spacecraft lifetime (2010). http://voyager.jpl.nasa.gov/spacecraft/spacecraftlife.html

[15] NASA JPL: Who uses AMMOS? http://ammos.jpl.nasa.gov/whousesammos/

[16] Needels, L. DSMS Information Systems Architecture (DISA) overview. Presented at: JPL, Pasadena, April 3, 2006. http://hdl.handle.net/2014/39174

[17] Object Management Group. Object Constraint Language, version 2.2. Document formal/2010-02-01, Object Management Group, Needham (2010). http://www.omg.org/spec/OCL/2.2/

[18] Object Management Group. OMG Systems Modeling Language (OMG SysML) specification. Adopted Specification ptc/06-05-04, Object Management Group, Needham (2006). http://www.sysml.org/docs/specs/OMGSysML-FAS-06-05-04.pdf

[19] Object Management Group. OMG Systems Modeling Language (OMG SysML), version 1.2. Document formal/2010-06-01, Object Management Group, Needham (2010). http://www.omg.org/spec/SysML/1.2/

[20] Object Management Group. UML for Systems Engineering: Request for Proposal. Document ad/03-03-41, Object Management Group, Needham (2003). http://www.omg.org/cgi-bin/doc?ad/2003-3-41

[21] Oussalah, M., Sadou, N., Tamzalit, D.: SAEV: A model to face evolution problem in software architecture. In: L. Duchien, M. D'Hondt, T. Mens (eds.) Proceedings of the International ERCIM Workshop on Software Evolution 2006, pp. 137–146. USTL, Lille (2006)

[22] Ozkaya, I., Diaz-Pace, A., Gurfinkel, A., Chaki, S.: Using architecturally significant requirements for guiding system evolution. In: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR 2010), pp. 127–136. IEEE, Los Alamitos (2010)

[23] Sanders, A. Innovation at JPL—GDS modernization: A case study. Presented at: Ground Systems Architecture Workshop (GSAW 2010), Manhattan Beach, CA, March 2, 2010. http://csse.usc.edu/gsaw/gsaw2010/s11b/sanders.pdf

[24] Sturdevant, K.F. Cruisin' and chillin': Testing the Java-based distributed ground data system "Chill" with CruiseControl. In: Proceedings of the IEEE Aerospace Conference. IEEE (2007)

[25] SysML Partners. Systems Modeling Language (SysML) Specification, version 1.0 alpha (2005). http://www.sysml.org/docs/specs/SysMLv1.0a-051114R1.pdf

[26] Wermelinger, M., Fiadeiro, J.L.: A graph transformation approach to software architecture reconfiguration. Sci. Comput. Program. **44**(2), 133–155 (2002)